

NetCAS: Dynamic Cache and Backend Device Management in Networked Environments

Joon Yong Hwang*

Cornelis Networks

Pennsylvania, USA

brian.hwang@cornelisnetworks.com

Chanseo Park*

Sungkyunkwan University

Department of Electrical and

Computer Engineering

Suwon, South Korea

cstim@g.skku.edu

Younghoon Kim†

Ajou University

Department of Software

Suwon, South Korea

yhoon@ajou.ac.kr

Abstract—Modern storage systems often combine fast cache with slower backend devices to accelerate I/O. As performance gaps narrow, concurrently accessing both devices, rather than relying solely on cache hits, can improve throughput. However, in data centers, remote backend storage accessed over networks suffers from unpredictable contention, complicating this split. We present NetCAS, a framework that dynamically splits I/O between cache and backend devices based on real-time network feedback and a precomputed Perf Profile. Unlike traditional hit-rate-based policies, NetCAS adapts split ratios to workload configuration and networking performance. NetCAS employs a low-overhead batched round-robin scheduler to enforce splits, avoiding per-request costs. It achieves up to 185% higher performance than traditional caching in remote storage environments and outperforms converging schemes like Orthus by up to 3.5× under fluctuating network conditions.

Index Terms—Networked storage, Caching systems, NVMe over Fabrics, Disaggregated storage, I/O scheduling

I. INTRODUCTION

Recent advances in storage technology have narrowed the performance gap between cache and backend storage. Whereas older hierarchies (e.g., HDD + SSD) showed clear asymmetry, modern pairings (e.g., NVMe + PMem) often exhibit overlapping throughput and latency. This trend has led to a range of systems that depart from cache-centric hierarchies and instead exploit parallel utilization of heterogeneous devices to maximize throughput. A growing body of work has shown that distributing I/O across tiers, rather than funneling all traffic through the cache, can harness aggregate bandwidth, avoid device-specific bottlenecks, and sustain performance at scale. Recent work in both file systems [1], [2], [3], [4] and caching frameworks [5], [6], [7], [8] demonstrates that parallel, nonexclusive access to multiple devices consistently outperforms strict hierarchical layering.

While prior efforts demonstrated the benefits of parallel utilization, they largely assumed local environments where cache and backend devices are co-located within the same

host. Modern datacenter architectures, however, increasingly rely on disaggregated storage [9], [10] designs where application servers access remote storage devices over networks. This separation introduces a key performance challenge: backend device performance becomes unstable and fluctuates due to network variability. Even with RDMA and advanced fabrics, remote I/O remains subject to congestion and interference that can inflate latency and reduce throughput unpredictably [11], [12], [13]. As a result, static or convergence-based I/O splitting strategies, which implicitly assume stable device performance, are insufficient in these environments, and adaptive network-aware approaches are needed to dynamically adjust caching and splitting decisions in response to observed network and device conditions.

This paper presents NetCAS, a network-aware caching and splitting framework that extends non-hierarchical caching into remote-storage environments. NetCAS addresses the limitations of the prior splitting approaches by introducing a precomputed Perf Profile based dynamic split model. For each workload configuration, NetCAS consults pre-profiled data to determine an ideal split ratio. At runtime, it monitors network performance in real time and adjusts the estimated backend device performance accordingly. This allows NetCAS to dynamically compute a new split ratio that reflects current network conditions.

Through extensive evaluation, we show that NetCAS:

- Achieves up to 185% performance improvement compared to traditional caching in remote storage environments.
- Outperforms hit-rate-based converging approaches like OrthusCAS by up to 3.5× under fluctuating network conditions.
- Introduces negligible CPU overhead by integrating directly into OpenCAS's fast I/O path and avoiding per-request decision-making outside existing control flow.

By bridging the gap between adaptive caching and dynamic network conditions, NetCAS offers a scalable and efficient

* These authors equally contributed to the work.

† Corresponding author.

solution for next-generation hybrid storage systems in modern datacenters.

II. BACKGROUND

A. Evolving Storage Devices

The traditional storage hierarchy was historically characterized by clear and stable performance asymmetry. Mechanically driven HDDs exhibited millisecond-scale latency and limited IOPS, while DRAM provided nanosecond latency and orders-of-magnitude higher bandwidth. This large and consistent gap justified strict hierarchical designs in which higher tiers absorbed nearly all performance-critical requests.

However, the storage landscape has changed rapidly over the past decade. Emerging technologies—including NVMe Flash SSDs, low-latency SSDs (e.g., Optane-class devices), persistent memory (PMem), and CXL-attached memory—have significantly reduced access latency and increased internal parallelism [14], [8], [15], [16], [17]. Although a rough ordering in terms of access latency still exists (e.g., DRAM < PMem < NVMe SSD < SATA SSD), bandwidth and IOPS no longer follow a strict total order across devices [14], [5], [15].

Recent empirical studies demonstrate that performance ratios between adjacent tiers vary substantially with workload characteristics and concurrency levels [5], [15], [14]. Under low concurrency, faster devices often dominate as expected. However, under higher parallelism, internal device parallelism and queueing behavior allow lower tiers to scale, narrowing or even eliminating the throughput gap [5], [14]. For example, Optane SSD and high-end NVMe Flash SSD can exhibit nearly identical throughput under sufficiently high thread counts. In some cases, write performance ordering may even invert depending on concurrency and access patterns [5], [8], [15].

These observations indicate the important trend. The modern storage hierarchy is no longer strictly hierarchical in performance; neighboring layers frequently exhibit overlapping throughput regions. Consequently, the assumption that a higher tier strictly dominates a lower tier does not consistently hold in contemporary systems. This shift fundamentally challenges traditional hierarchical caching strategies that attempt to maximize hit rate at the fastest device.

B. NVMe and NVMe over Fabrics

NVMe is a high-performance interface designed to exploit the parallelism of non-volatile memory devices over PCIe [18]. Its queue-based architecture supports multiple submission and completion queues, enabling lock-free multi-core scaling and minimizing synchronization overhead [18]. By allowing each CPU core to interact with independent queue pairs, NVMe reduces contention in the I/O path and sustains high IOPS and bandwidth under concurrency [18], [14].

NVMe over Fabrics (NVMe-oF) extends NVMe semantics to network transports, most notably RDMA and TCP [19], [20], [21]. Among these, RDMA-based NVMe-oF is particularly significant for high-performance deployments [20]. RDMA enables zero-copy data transfer, kernel bypass, and

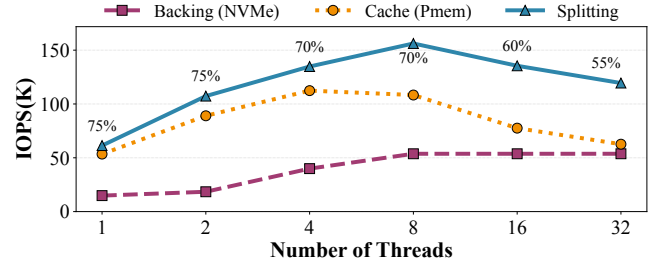


Fig. 1. Throughput comparison between cache device (PMem), backend device (NVMe), and splitting at the optimal ratio across varying thread counts. The percentage labels on the splitting line denote the optimal split ratio at each concurrency (e.g., 75% indicates 75% of requests sent to cache and 25% to backend).

direct memory access between hosts and remote storage targets, thereby eliminating intermediate buffer copies and reducing CPU involvement [20]. Completion events are generated directly by the NIC, and data movement is offloaded to hardware engines, allowing NVMe command submission and completion semantics to be preserved across the network. Prior characterizations show that transport and network conditions still materially affect realized throughput and latency in disaggregated deployments [22], [13], [23].

C. OpenCAS and Hybrid Storage Abstraction

OpenCAS (Open Cache Acceleration Software) [24] is an Intel-initiated open-source caching project that is now community maintained. It provides a complete block-caching stack—libraries, adapters, and tools—whose goal is to accelerate a backend block device by leveraging a higher-performance cache device. At its core is the Open CAS Framework (OCF), a high-performance caching meta-library that supports configurable caching policies. In this paper we focus on Open CAS Linux, the kernel block-layer integration used in our prototype. OpenCAS is device-agnostic: it can pair diverse cache/backend device types, including NVMe SSDs, PMem, SATA SSDs, and HDDs, while preserving a single logical volume and policy-driven routing.

As OpenCAS operates entirely within the kernel I/O path and maintains a unified fast path, it provides a practical foundation for implementing adaptive request-splitting mechanisms without modifying upper software layers.

D. Non-Hierarchical Caching (NHC)

Non-hierarchical caching (NHC), as exemplified by Orthus-style designs [5], [15], departs from hit-rate-maximizing policies by allowing concurrent utilization of both cache and backend devices. Traditional hierarchical caching pushes for near-100% hits: hot data are copied into the fast device, misses are promoted, and the cache tier is treated as the primary performance engine. This is appropriate when the cache is overwhelmingly faster and the backend contributes little beyond capacity. However, once the cache saturates, throughput is capped at the cache device's peak and additional

requests only add queueing; meanwhile, backend bandwidth sits idle even though it could serve useful work. This mismatch becomes more visible in modern hierarchies where device throughput curves overlap under high concurrency.

NHC treats both devices as independent contributors to total throughput and explicitly splits requests across them, even sending a fraction of cache hits to the backend when it helps. By exploiting aggregate bandwidth, performance can exceed the cache-only ceiling and approach the sum of both devices' usable throughput when the workload and concurrency permit. In effect, NHC optimizes end-to-end throughput rather than hit ratio alone, and avoids unnecessary promotions that only add traffic to an already saturated cache.

To validate this, we perform a simple experiment: for a fixed concurrency level, we sweep the read split ratio between cache and backend (from 100% cache to 100% backend) and record aggregate throughput at each point. We then compare the best-performing split against vanilla OpenCAS, which serves reads exclusively from the cache device, and against the backend device used alone. Figure 1 shows the results for our cache/backend pair (PMem/NVMe). At every concurrency level, the best split exceeds both the cache-only and backend-only baselines, confirming that parallel utilization unlocks bandwidth that neither device can deliver alone. At lower thread counts, assigning most requests to the cache is effective, but as parallelism increases the backend contributes more useful throughput and the optimal ratio shifts accordingly. This trend is consistent with prior NHC observations that throughput-optimal policies are workload- and concurrency-dependent rather than fixed [5].

III. MOTIVATION

A. Disaggregation Trends in Modern Datacenters

Modern datacenters are increasingly shifting from server-attached disks toward *disaggregated storage*, where compute and storage scale independently across the network. This shift improves hardware utilization, elasticity, and cost efficiency [25], [26], [9], while centralized pools simplify management and enable multi-tenancy; major cloud systems such as AWS Aurora [9] and Azure SQL Hyperscale [10] already adopt this model at scale.

As discussed in §II-B, *NVMe over Fabrics (NVMe-oF)* [19] preserves key NVMe advantages even over the network, making remote NVMe practical for latency-sensitive services. Together with high-speed fabrics [23], [27], this means remote storage can no longer be treated as an exceptionally slow tier; instead, disaggregated deployments become a realistic operating point where split policies must adapt to network variability.

B. Limits of Hierarchical Caching

As described in §II-D, modern cache/backend pairs often have overlapping throughput regions, so maximizing hit rate is not equivalent to maximizing throughput. Figure 1 further shows that split routing can outperform cache-only routing and that the best split changes with concurrency. Therefore,

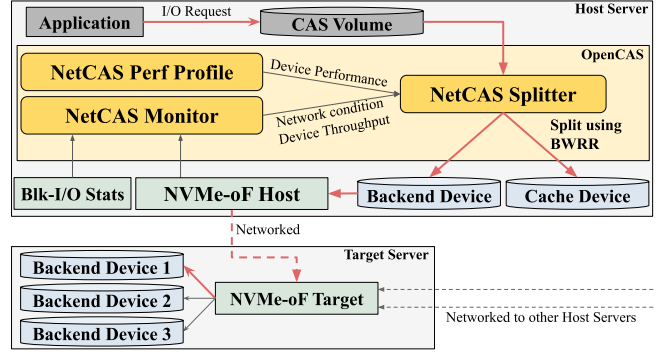


Fig. 2. NetCAS framework overview. Real time metrics from NetCAS Monitor and device baseline performance from NetCAS Perf Profile are passed to NetCAS Splitter, where I/O requests are dynamically routed to the local cache and remote backend device.

in our setting the motivation is not to further optimize hit-rate policies, but to build a controller that continuously tracks the throughput-optimal split point.

C. Challenges in Networked Environments

Given the Background observations above, the remaining gap is operational: the optimal split must be maintained under network dynamics in real deployments. In disaggregated environments, this creates the following concrete challenges:

- (i) **Throughput Variability.** Backend capacity fluctuates with network congestion and cross-traffic. A split ratio computed under one condition may quickly become suboptimal as network state changes.
- (ii) **Congestion Amplification.** Over-directing requests to a temporarily degraded backend can cause queue buildup and latency inflation, which further reduces effective throughput.
- (iii) **Imbalanced Dispatch.** Even with a correct long-term split ratio, bursty or uneven dispatch can cause transient device idling or saturation, reducing aggregate efficiency.
- (iv) **Estimation Lag.** Profiling-based or convergence-based schemes react slowly to rapid network changes, leading to persistent mismatch between assumed and actual backend performance.

Together, these challenges motivate NetCAS as a lightweight network-aware controller that preserves NHC benefits under time-varying backend conditions.

IV. DESIGN AND IMPLEMENTATION

A. Framework Requirements and Design

Maximizing throughput in hybrid storage requires a framework that continuously adapts to device performance while remaining lightweight and transparent. NetCAS is designed to satisfy four such requirements in a unified architecture, which is depicted in Figure 2.

First, real-time performance detection. Device throughput can fluctuate significantly, especially when backend devices

are remote and subject to network contention. Without timely visibility into these changes, request distribution risks either overloading a degraded device or underutilizing an available one. To enable real-time performance detection, network condition for this work, we patch the NVMe-oF host kernel module to measure fabric throughput and latency as requests complete (§IV-B). By instrumenting its request-completion path, NetCAS obtains accurate, low-overhead metrics that reflect instantaneous network conditions. These metrics feed directly into the NetCAS congestion detector (§IV-D), which quantifies bandwidth loss and latency inflation into a unified severity score for adaptation.

Second, adaptive splitting. The system should adjust the split ratio in real time. Static ratios or slow convergence are inadequate when device performance or network conditions change rapidly. NetCAS relies on a precomputed Perf Profile (§IV-C) that stores empirically optimal ratios for different workload configurations, indexed by parameters. Using these parameters, NetCAS computes the ratio using the analytical model (§IV-E), ensuring consistency between profiling and online adaptation.

Third, application transparency. Splitting logic must operate without requiring applications or file systems to change their behavior, and must remain independent of specific caching policies (e.g., write-back, write-through). NetCAS achieves this by extending OpenCAS. OpenCAS [24] is an open-source block-level caching system that unifies a fast cache device with a slower backend device under a single logical volume, while supporting multiple caching policies. By extending OpenCAS, NetCAS can split requests across devices without altering higher-level semantics (§IV-H).

Fourth, low overhead and high performance. Additional threads, locks, or context switches could undermine throughput gains. NetCAS executes all scheduling inline with lightweight logic, avoiding extra threads or locks. Furthermore, coarse-grained request distribution, even at the correct ratio, may still stall one device while the other idles. NetCAS therefore employs Batched Weighted Round Robin (BWRR) Scheduler (§IV-F) to interleave requests, preventing blocking and keeping both cache and backend busy under high concurrency.

B. Measuring Performance Fluctuations

In local environments, devices exhibit relatively stable performance that is easy to measure. In contrast, when backend devices are accessed over the network, fluctuations arise from congestion and resource contention, making measurement more challenging. One possible approach is a centralized controller that continuously collects global network state and allocates backend bandwidth across hosts. We intentionally avoid this design: it requires intrusive cross-node signaling, policy coupling with cluster schedulers, and tight control-plane synchronization, all of which increase deployment complexity and reaction latency. Instead, NetCAS follows an end-host design philosophy and detects performance degradation from host-local observations.

The dominant source of variability is the network transport itself, so NetCAS instruments the NVMe-oF RDMA host kernel module to monitor throughput and latency as I/O requests complete. Unlike coarse network counters, these metrics are *storage-specific*: they reflect the actual performance of the remote device path, isolated from unrelated application traffic. These throughput and latency measurements are the *sole* inputs to the congestion detector (§IV-D), which computes the severity score used to adjust the split ratio.

In addition to fabric metrics, NetCAS Monitor leverages block-layer I/O counters exposed via sysfs to derive device throughput. These counters serve a complementary role: rather than feeding the congestion detector, they drive I/O detection and mode transitions (§IV-H), determining *when* to activate or deactivate splitting, but are *not* used for congestion detection or split-ratio computation.

Synchronizing these heterogeneous signals on a common sampling interval ensures that the splitter bases its decisions on both network and device conditions, while providing a modular architecture where new monitors can be integrated without restructuring the system. While our current implementation focuses on network-induced variability, the same host-local framework naturally extends to other performance-shift sources (e.g., device-level interference or firmware throttling) by adding new metric sources.

A natural concern is whether NetCAS must distinguish network congestion from storage-side congestion—for example, a shared storage target stuttering under garbage collection or multi-tenant load. In practice, the distinction is unnecessary: from a host’s perspective, any backend slowdown appears as reduced end-to-end throughput and increased completion latency. Because NetCAS reacts to these end-to-end signals rather than diagnosing root cause, the response remains consistent regardless of origin: shift more requests to the local cache and reduce backend share. Once slowdown clears, whether network- or storage-induced, the profile-based ratio is restored immediately, avoiding the slow additive recovery of convergence-based schemes.

Therefore, NetCAS requires no cross-node coordination protocol. Each host independently observes its own path conditions and converges to a split ratio that reflects its instantaneous fair share of backend bandwidth. This host-autonomous design keeps the data path simple, scales with cluster size, and remains deployable incrementally without introducing a global control dependency.

C. Performance Profile

To split requests efficiently without incurring costly online exploration, NetCAS relies on a **Performance Profile (Perf Profile)** that empirically records standalone throughputs of cache and backend devices for different workload configurations. The profile spans a three-dimensional space defined by *block size*, *in-flight requests*, and *threads*. These values are later referenced when determining the optimal split ratio. Such profile-based approaches are widely used in systems for rapid decision making, as they capture device-specific scaling

behaviors while enabling constant-time lookups in the fast path. By consulting a compact profile, they avoid latency and CPU overhead in storage and networking systems such as congestion-control protocols [28], DVFS power management [29], and TCP CUBIC’s cubic-root calculation [30].

Internally, the Perf Profile is a lookup table (LUT) indexed by $\langle \text{block_size}, \text{inflight_requests}, \text{threads} \rangle$, where each entry stores the independently measured standalone throughput of the cache device and the backend device at that operating point. The initial grid spans 5 inflight levels $\times$ 5 thread levels $\times$ 2 block sizes = 50 entries, with concurrency levels drawn from commonly exercised datacenter settings and block sizes matching the most common OpenCAS page sizes.

These entries are populated with a *generic I/O microbenchmark* (e.g., `fio` with random reads) rather than an application-specific workload, because the LUT captures *device-level concurrency characteristics*—a hardware property of each device pair—that transfer across workloads sharing the same hardware. This is analogous to how transport-layer congestion-control profiles characterize link capacity rather than application behavior; modern datacenter environments further justify profile reuse, as hardware and workload mixes within a given cluster are highly homogeneous [31], [32]. Industry practice confirms this pattern: Meta deploys uniform 48-rack server pods as standardized building blocks [33], Google organizes dedicated TPU Pod clusters of identical accelerators [34], Azure assembles thousands of identical GPU servers into AI supercomputer clusters [35], and Alibaba’s scheduling constraints confine most jobs to a small subset of eligible nodes [36].

When the runtime workload operates between grid points, NetCAS uses the *nearest LUT entry* as a starting estimate. The online monitor (§IV-B) immediately observes actual throughput and feeds the congestion detector, which recalculates the split ratio using the formula in §IV-E. The LUT thus provides a *fast starting point*; the online monitor dominates within seconds. New entries can also be appended at runtime as the monitor encounters previously unseen workload configurations, making the profile incrementally self-improving.

Constructing the full table can be costly. In modern datacenters, however, hardware and workload mixes are relatively homogeneous, and many services operate under stable, repeatable configurations. This lets operators prebuild a profile for a fixed workload or maintain a shared profile that can be reused across servers, avoiding per-machine exploration. Figure 3 quantifies this trade-off. The 50-entry LUT completes in approximately 25 minutes, after which NetCAS operates at its steady-state split ratio. The cumulative gain crosses zero (break-even) at 59 minutes and continues to grow, reaching +49% at three hours and asymptotically approaching the +73% steady-state advantage. Since modern datacenter jobs typically run for hours to days, the profiling cost is amortized quickly.

D. Detecting Congestion

We detect fabric anomalies using a sliding RDMA window over completed I/O to make the NetCAS detector robust to

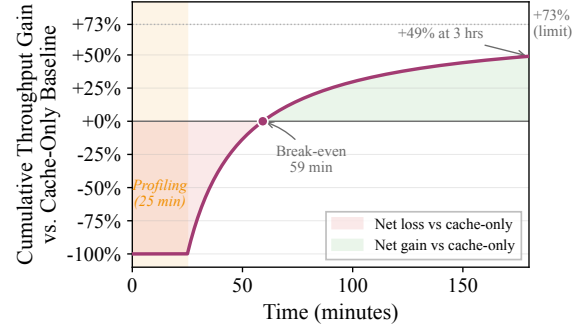


Fig. 3. Cumulative throughput gain of NetCAS over a cache-only baseline (inflight requests = 16, threads = 16). During the one-time profiling phase (25 min), the system populates the LUT and operates at reduced throughput, creating an initial deficit. The break-even point is reached at 59 min; beyond that, every additional minute of operation adds net gain, approaching the +73% steady-state limit. At the 3-hour mark cumulative gain stands at +49%.

transient bursts and queuing noise. Every monitoring epoch, the NetCAS monitor exports per-epoch throughput B_t and latency L_t . The NetCAS detector maintains baselines given by the maximum observed throughput $\bar{B}$ and the minimum observed latency $\bar{L}$, and computes normalized deviations

$$\delta_B = \frac{\bar{B} - B_t}{\bar{B}}, \quad \delta_L = \frac{L_t - \bar{L}}{\bar{L}}.$$

From these deviations we compute a single severity score

$$\text{drop_permil} = 1000 \cdot (\beta_B \delta_B + \beta_L \delta_L),$$

where the weights β_B and β_L control which signal is emphasized (set to $\beta_B = \beta_L = 0.5$ in our prototype to treat bandwidth degradation and latency inflation equally). The result, denoted `drop_permil` (a per-thousand penalty factor), provides a joint view of bandwidth loss and latency inflation and is used to calculate the optimal split ratio under network congestion. Note that our implementation can easily be extended to cache-device congestion by exposing block-layer I/O counters for the cache device without architectural changes as previously mentioned.

E. Calculating the Split Ratio

While the Perf Profile records device performances, we also need a principled way to calculate the *ideal* split. Prior work [5] showed that when many requests are issued in parallel, completion time can be modeled by balancing the service times of each device. With a fraction r of requests sent to the cache and $1 - r$ to the backend, the per-device service times are

$$T_{\text{cache}} = \frac{r}{I_{\text{cache}}}, \quad T_{\text{back}} = \frac{1-r}{I_{\text{back}}},$$

where I_{cache} and I_{back} are standalone throughputs from Perf Profile. The batch completes only when the slower side finishes:

$$T_{\text{total}} = \max\left(\frac{r}{I_{\text{cache}}}, \frac{1-r}{I_{\text{back}}}\right).$$

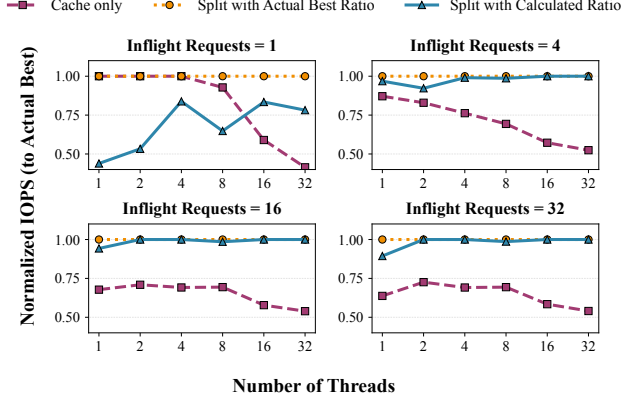


Fig. 4. Normalized throughput under different inflight request counts without network congestion. At low concurrency the calculated split deviates from the empirical best, but accuracy improves quickly with higher concurrency, converging to the optimal ratio.

The minimizer of T_{total} lies at the intersection of the two, yielding ρ_{base} , the optimal split ratio without network congestion,

$$\rho_{\text{base}} = \frac{I_{\text{cache}}}{I_{\text{cache}} + I_{\text{back}}}.$$

When congestion is detected, the splitter applies the observed drop_perml $d \in [0, 1000]$ to scale down the backend throughput, recomputing

$$\rho = \frac{I_{\text{cache}}}{I_{\text{cache}} + I_{\text{back}} \cdot (1 - d/1000)},$$

so that the ratio adapts smoothly to degraded conditions.

In practice the model is inaccurate at very low queue depths: with only one or two in-flight requests, a single operation directed to the slower device blocks completion and variance tends to dominate. As concurrency increases, as is typical in datacenter workloads [37], [14], [38], both devices stay busy in parallel and the prediction rapidly converges to measured throughput (Figure 4).

F. Selecting the Device

To enforce the split ratio ρ in practice, NetCAS employs a **Batched Weighted Round Robin (BWRR)** scheduler. Whereas the Perf Profile stores per-device throughput for each workload—providing *macroscopic* guidance by encoding the optimal long-term ratio—BWRR delivers *microscopic* control at the request level so the desired ratio is realized in short windows. This prevents burstiness, avoids idle slots on either device, and keeps both cache and backend continuously utilized. Algorithm 1 shows the core logic.

BWRR combines three mechanisms: (i) enforcing long-term ratios by expected counts, (ii) maintaining the ratios even within short-term intervals with minimal repeating pattern (via GCD), and (iii) filling residual imbalance with quota-based dispatch. This multi-tiered control keeps the ratio accurate even at small window sizes while avoiding bursts or starvation. As shown in the ablation study (Fig. 5),

Algorithm 1 Batched Weighted Round Robin (BWRR)

Require: split_ratio ρ , window_size W , batch_size B

```

1: procedure SENDCACHE( $r$ )
2:   cache_quota  $\leftarrow$  cache_quota  $-1$ 
3:   Send  $r$  to CACHE
4: procedure SENDBACK( $r$ )
5:   backend_quota  $\leftarrow$  backend_quota  $-1$ 
6:   Send  $r$  to BACKEND
7: for each incoming req  $r$  do
8:   if req_count =  $W$  then
9:      $a \leftarrow \text{round}(\rho W)$ ;  $b \leftarrow W - a$ 
10:    pattern_size  $\leftarrow \min(W/\text{gcd}(a, b), B)$ 
11:    pattern_cache  $\leftarrow \lfloor (\text{pattern\_size} \cdot a)/W \rfloor$ 
12:    pos  $\leftarrow 0$ ; req_count  $\leftarrow 0$ 
13:    cache_quota  $\leftarrow a$ ; backend_quota  $\leftarrow b$ 
14:   if cache_quota > 0 and backend_quota > 0 then
15:     if pos > pattern_cache then SENDBACK( $r$ )
16:     else SENDCACHE( $r$ )
17:     pos  $\leftarrow (\text{pos} + 1) \bmod \text{pattern\_size}$ 
18:   else if cache_quota = 0 then SENDBACK( $r$ )
19:   else if backend_quota = 0 then SENDCACHE( $r$ )
20:   req_count  $\leftarrow$  req_count + 1

```

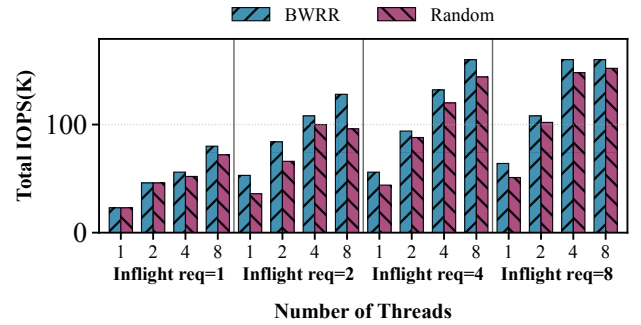


Fig. 5. Throughput comparison of BWRR versus random dispatch across inflight requests and thread counts. BWRR sustains the target ratio more evenly, delivering higher aggregate IOPS especially under shallow queues where randomization causes imbalance.

BWRR maintains the target ratio far more evenly than random dispatch, yielding higher aggregate throughput under shallow queues where randomization wastes parallelism.

Concretely, consider a window of $W=10$ requests with $\rho=0.7$. BWRR assigns $a=7$ slots to cache and $b=3$ to backend. Because $\text{gcd}(7, 3)=1$, the minimal repeating pattern spans all 10 slots: the first 7 go to cache, the next 3 to backend, and the cycle restarts. As each request arrives, BWRR advances through this interleaved pattern, guaranteeing that the target ratio is maintained even within short bursts. If one device's quota exhausts before the window closes, all remaining requests go to the other device, ensuring exact adherence to ρ over every window.

The batch parameter B caps the pattern length so that both

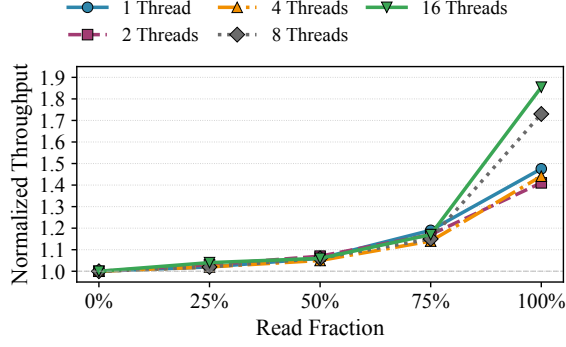


Fig. 6. Throughput under varying read/write ratios without network congestion (NT=16 inflights). Gains scale roughly linearly with the read fraction but fall slightly short due to write-side contention on the cache device.

devices remain active when a and b share no common factor. For example, with $W=100$ and $\rho=0.17$ we get $a=17$, $b=83$, $\gcd(a,b)=1$; without B , the pattern would span the entire window and serialize 17 cache requests before 83 backend requests, leaving one device idle. Setting $B=10$ truncates the pattern to 10 slots (1 cache, 9 backend), keeping both devices busy in parallel. Any rounding residue is reconciled at the window boundary by the per-device quotas (lines 13, 18–19), which redirect remaining requests once a quota is exhausted—guaranteeing the window-level split equals ρ .

G. Read/Write Mixed Workload Behavior

Importantly, NetCAS does not modify, intercept, or reorder any write operation. All splitting decisions apply exclusively to *cache-hit read* requests; write allocation, eviction ordering, and durability guarantees remain exactly as implemented by the underlying OpenCAS cache mode (write-back or write-through). This boundary is a deliberate design choice: modifying the write path would require dependency tracking or journaling to preserve consistency [39], and prior work on non-hierarchical caching likewise keeps write allocation unchanged for endurance and persistence [5]. By restricting splitting to the read side, NetCAS inherits OpenCAS’s proven consistency model while avoiding write-path coordination entirely.

Because NetCAS only splits cache-hit reads, its benefit scales roughly linearly with the read fraction of the workload. Figure 6 shows this trend across all thread counts: as the read share grows, throughput increases in near-proportion.

The gains fall just short of perfect linear scaling at intermediate read fractions. The gap is attributable to write traffic: in write-through mode every write must still be served exclusively by the cache device, so the cache queue never fully drains regardless of how many reads are offloaded to the backend. This residual write contention prevents the throughput improvement from growing in exact lockstep with the read ratio. At high thread counts (8 and 16 threads) the pure-read case reaches $1.73\times$ and $1.85\times$ respectively, confirming that NetCAS’s benefit grows with concurrency as the splitter keeps both devices saturated.

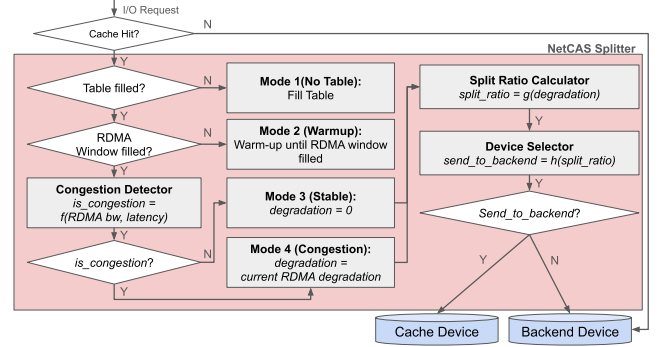


Fig. 7. NetCAS mode-transition state machine. Arrows denote event-driven transitions: LUT population completes (*No Table*→*Warmup*), monitoring baselines stabilize (*Warmup*→*Stable*), the congestion detector fires (*Stable*→*Congestion*), and fabric metrics recover (*Congestion*→*Stable*). In *Stable* mode the splitter uses the LUT-derived ratio with near-zero overhead; in *Congestion* mode the ratio is recalculated every epoch from live network metrics.

H. Integration and Footprint

We implement splitting directly above `engine_fast()` in OpenCAS, the unified fast path for handling cache hits across write policies. NetCAS injects NetCAS splitter here to decide whether a request is served by the cache or redirected to the backend device; misses always go to the backend device, keeping the design safe and policy-agnostic. Since allocation, completion, and locking are already handled in upper layers, the NetCAS splitter hooks into the mid-path without new threads or locks, preserving transparency while avoiding extra synchronization.

To minimize overhead while adapting to network dynamics, NetCAS employs a lightweight mode-based control scheme as depicted in Figure 7: **No Table** populates the Perf Profile, **Warmup** stabilizes monitoring windows, **Stable** applies pre-computed ratios with near-zero cost, and **Congestion** periodically recalculates ratios to reconfigure BWRR. Transitions are event-driven: once the LUT is fully populated for the current workload class, the system moves from *No Table* to *Warmup*; after the monitoring windows accumulate enough samples for a statistically stable baseline, *Warmup* advances to *Stable*. In *Stable* mode the splitter serves requests at the LUT-derived ratio with near-zero overhead. When the congestion detector signals a sustained throughput drop or latency spike on the fabric (Section §IV-D), the system enters *Congestion* mode and recalculates the split ratio every epoch using live network metrics. Once fabric performance recovers, the ratio reverts to the profile-based value and the system returns to *Stable*. By confining calibration and monitoring to transient phases, the NetCAS splitter stays responsive to change yet incurs virtually no steady-state overhead. Even under heavy workloads (16 threads $\times$ 16 inflight requests), the total utilization of NetCAS rises only from 12.46% (OpenCAS) to 12.79%, a negligible 0.33% absolute difference.

NetCAS required modest kernel modifications with less than 1,200 LOC modified to the OpenCAS and NVMe-oF RDMA

kernel module. The full source code and experimental artifacts are available at: <https://github.com/NetCAS-SKKU/NetCAS>.

V. EVALUATION

A. Evaluation Setup

Testbed. We evaluate NetCAS on a dual-socket Intel Xeon Gold 6330 server (56 cores total, 2.0 GHz) with 384 GB DRAM running Linux 5.15 and OpenCAS v22.3. OpenCAS is configured in *write-through* mode so that every write is committed to the backend device synchronously; this eliminates write-ordering concerns and isolates the effect of read-path splitting. The cache device is a local Intel Optane Persistent Memory module, while the backend device is a remote Samsung 990 Pro NVMe SSD accessed over NVMe-oF (RDMA) through a Mellanox ConnectX-5 100 Gbps NIC. The network topology consists of three host servers connected to one target storage server through a middle switch: the target uses a 40 Gbps NIC, while the hosts and switch use 100 Gbps NICs, creating a single congestion point at the target.

Workloads and Baselines. Synthetic workloads are generated with `fio (bs=64k,ioengine=libaio,direct=1,size=1G)`, sweeping I/O depth and thread count. For real-world application behavior, we use Filebench workloads as described in §V-E, covering random-read, sequential-read, and mixed read/write patterns. Network congestion is injected with `ib_write_bw` using 1 MB messages, queue depth 1, rate-limited to 2.5 Gb/s. Before each run the cache is warmed: a sequential write fills the device, followed by 120 s of random writes; the cache is then flushed and statistics reset. We compare NetCAS against three baselines: vanilla OpenCAS (cache standalone), backend standalone, and the state-of-the-art OrthusCAS [5]. OrthusCAS converges its split ratio by monitoring per-device block-layer throughput statistics. Because Intel Optane PMem does not expose these counters, the convergence loop cannot operate; the ratio only re-adjusts when the cache hit rate changes. We therefore supply OrthusCAS with the empirically best static ratio for each concurrency level, giving it an upper-bound advantage that a live deployment would not achieve.

All experiments use a prefilled, prewarmed cache so that every read hits the cache device. This isolates the splitting mechanism: since NetCAS only modifies the read path for cache-hit requests (§IV-G), prefilling ensures every I/O exercises the splitter. Sections §V-B, §V-C use read-only workloads to measure the maximum splitting benefit; Section §V-E then introduces mixed read/write ratios to confirm that gains scale with the read fraction and that the unmodified write path introduces no regressions, consistent with the trends reported by OrthusCAS [5].

B. Baseline Performance

We begin by comparing NetCAS with vanilla OpenCAS and OrthusCAS under contention-free conditions. Figure 8 reports aggregate throughput across concurrency levels. As expected, both NetCAS and OrthusCAS surpass vanilla OpenCAS by exploiting request splitting in most cases. When the

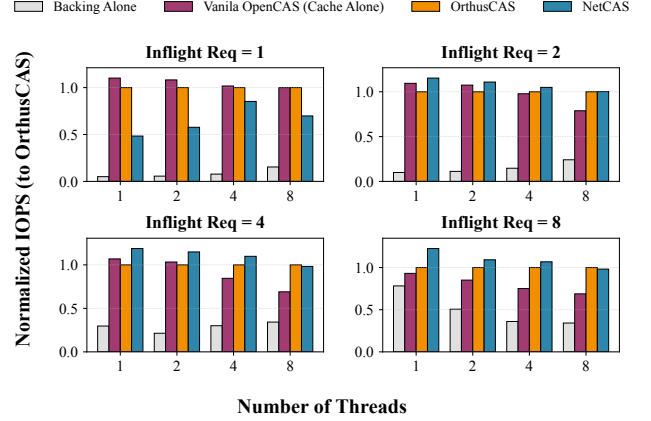


Fig. 8. Baseline throughput without contention. NetCAS achieves up to 125% higher throughput than OrthusCAS and 142% over vanilla OpenCAS across concurrency levels.

number of inflight requests is one, NetCAS falls short of OrthusCAS because the analytical split ratio deviates from the empirical optimum supplied to OrthusCAS. However, by operating without extra threads or locks, NetCAS outperforms OrthusCAS in most other regimes. At very low I/O depth (1–2 in-flight requests), network round-trip variance dominates and a single mis-routed request can stall the pipeline, limiting the benefit of any splitting scheme. As concurrency grows, pipelining amortizes this variance and NetCAS’s analytical ratio converges to the empirical optimum, explaining the widening gap in Figure 8.

C. Performance Under Contention

Next, we evaluate robustness under dynamic congestion. Figure 9 shows throughput over time under injected load. In Figure 9(a) and (b), both NetCAS and OrthusCAS outperform vanilla OpenCAS by exploiting split I/O, but when RDMA bandwidth is constrained, OrthusCAS drops sharply whereas NetCAS sustains higher throughput by rebalancing online. NetCAS achieves up to $3.5\times$ higher throughput than OrthusCAS at low-thread concurrency and still improves by about $1.2\times$ at high-thread concurrency.

Figure 9(c) uses a mixed read/write `fio` workload (same thread and I/O-depth setting as (b)). Compared with (a) and (b), panel (c) with mixed read/write workload has lower aggregate throughput than read-only cases, so 10 competing flows were insufficient to induce clear contention. We therefore increased contention intensity to 40 flows each, with 100s timeline providing enough window to spawn all 40 competing flows and capture both phases consistently.

In the mixed read/write workload, we observe a similar trend as in the read-only cases. For vanilla OpenCAS, although write-through semantics imply that congestion should impact both reads and writes, the overall bandwidth requirement of writes in this workload is relatively low; thus, the performance degradation is dominated by reads, which are more bandwidth-

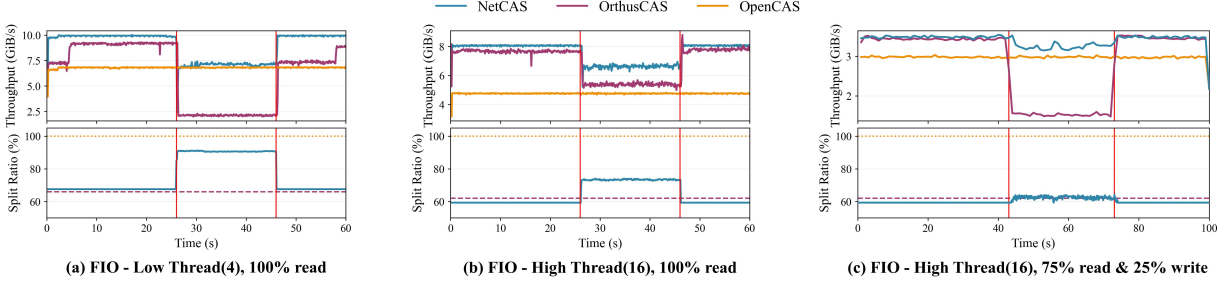


Fig. 9. Throughput under injected congestion. Panels (a) and (b) use read-only `fio` with 4 and 16 threads (16 in-flight requests), with a 60 s run and a 20 s contention window from 10 competing flows (2 servers, 2.5 Gbps cap per flow). Panel (c) uses mixed read/write `fio` with 16 threads and 16 in-flight requests, with a 100 s run and a 30 s contention window from 40 competing flows (2 servers, 2.5 Gbps cap per flow).

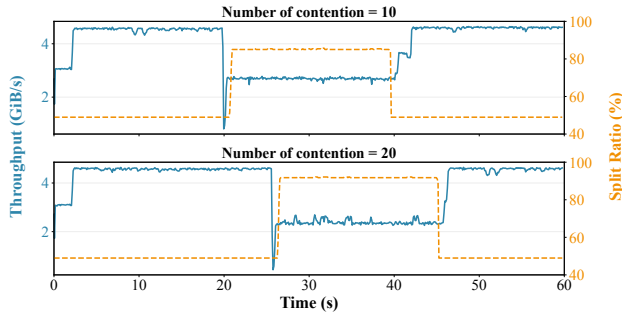


Fig. 10. Throughput under low and high contention for the same workload (inflight requests = 16, threads = 16). Each competing flow attempts to maximize its bandwidth without capping. NetCAS allocates a larger share to the cache as backend bandwidth is constrained, mitigating throughput loss.

intensive. As a result, both OrthusCAS and NetCAS exhibit more noticeable drops during congestion. However, NetCAS consistently maintains higher throughput. An interesting observation is that even when NetCAS dynamically adjusts its split ratio, the resulting ratio is similar to that of OrthusCAS, yet the achieved performance is significantly higher. This indicates that split ratio alone does not fully explain performance differences under mixed workloads. We attribute this gap primarily to metadata overhead. In NetCAS, read operations follow a pass-through write path that avoids metadata updates on the cache device for core-device accesses, whereas OrthusCAS updates metadata even without promoting data. Prior work has shown that metadata updates introduce non-trivial overhead [15]. Because reads are more sensitive to bandwidth under congestion, this additional metadata overhead disproportionately affects OrthusCAS, leading to lower overall throughput compared to NetCAS.

D. Performance Across Contention Levels

Since networked storage must serve workloads under diverse levels of background competition, we conduct an experiment to evaluate NetCAS across different contention loads (i.e., varying numbers of competing flows). Figure 10 shows the resulting throughput. As backend bandwidth is

squeezed by more contenders, NetCAS splitter raises the cache share, defending overall throughput without abrupt shifts. This smooth rebalancing demonstrates NetCAS's ability to scale protection gracefully under both light and heavy competition, rather than oscillating between extremes. This adaptability is especially relevant in datacenter networks where tenant bandwidth demands vary dynamically. By adjusting smoothly across contention levels, NetCAS avoids cliff effects and delivers high and predictable performance. At the same time, it preserves fairness: instead of monopolizing backend bandwidth, it stabilizes throughput by shifting excess load to the cache while respecting each flow's fabric share.

E. Performance With Real-World Workloads

To evaluate NetCAS under realistic application behavior, we use *Filebench* [40], [41], an application-level storage benchmark. Filebench generates workloads at the file-system interface, capturing key characteristics of real applications such as access locality, concurrency, and mixed I/O behavior.

We construct three representative workloads over a 10 GB dataset (1000 files, each 10 MB). Before each workload run, we apply the same warmup procedure to preload cache state for a consistent starting condition. All workloads bypass the page cache using `directio` for reads to directly exercise the cache-backend split path.

Workload A consists of 16 concurrent reader threads issuing 64 KB random reads over the dataset, representing a cache-friendly scenario that isolates cache-hit performance.

Workload B uses 16 threads performing sequential reads with 1 MB I/O size, scanning entire files. This models streaming or scan-heavy applications (e.g., analytics), where temporal locality is minimal and caching benefits are limited.

Workload C is a mixed workload with 16 reader threads and 2 writer threads. Readers perform 64 KB random reads with `directio`, while writers issue buffered random writes to maintain system stability. This setup emulates a read-dominant service workload with concurrent read/write interference.

Figure 11 shows throughput across these workloads, with and without contention. Under stable conditions, NetCAS consistently achieves the highest throughput, with the largest

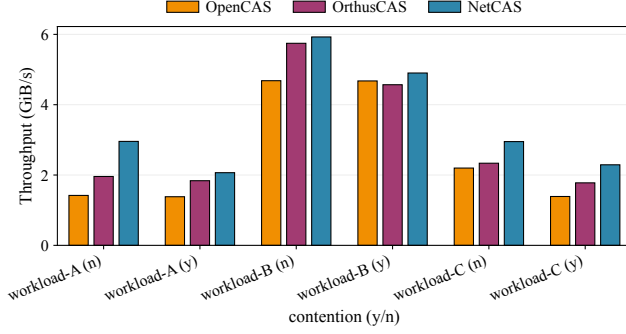


Fig. 11. Throughput of Filebench workloads A–C. Labels (n) and (y) indicate without and with contention, respectively; contested runs use 40 flows (two servers, 2.5Gbps cap per flow).

gain in Workload A, $2.1\times$ over OpenCAS and $1.5\times$ over OrthusCAS, due to efficient cache-hit handling. For Workload C, which mixes reads and writes, even vanilla OpenCAS sees a 37% throughput drop under contention: writes are pass-through operations that must traverse the congested network path regardless of caching policy. However, NetCAS sustains the highest throughput across all workloads, $1.65\times$ over OpenCAS and $1.29\times$ over OrthusCAS on Workload C under contention. In particular, OrthusCAS is more affected by degraded backend performance, while NetCAS adapts its split to mitigate the impact.

Figure 12 shows time-series throughput for Workload B under a 30 s congestion interval. OpenCAS serves all cache-hit reads from the local device and is therefore unaffected by network congestion, but it remains at a lower throughput ceiling because it cannot leverage the backend’s additional bandwidth (NetCAS achieves $1.27\times$ OpenCAS throughput in steady state). Once congestion begins, OrthusCAS exhibits pronounced throughput fluctuations: its static split ratio continues directing a fixed fraction of reads to the now-congested backend, causing a 20% throughput drop that pushes it below even vanilla OpenCAS. NetCAS adaptively shifts its ratio toward the cache, limiting its throughput reduction to 17% and sustaining $1.07\times$ the throughput of OrthusCAS during the congestion window, and quickly recovers after congestion clears, restoring steady-state performance within a short period. These results confirm that NetCAS’s adaptive splitting remains effective under realistic file-system workloads and dynamic network conditions.

VI. CONCLUSION

This paper presented NetCAS, a lightweight yet practical framework for hybrid storage in disaggregated environments, where cache and backend devices should be used *concurrently* rather than strictly hierarchically. The key challenge in this setting is that backend performance is no longer stable: even with NVMe-oF and RDMA, shared-fabric contention can rapidly shift the throughput-optimal split point. NetCAS addresses this by combining three components in a unified fast path: (i)

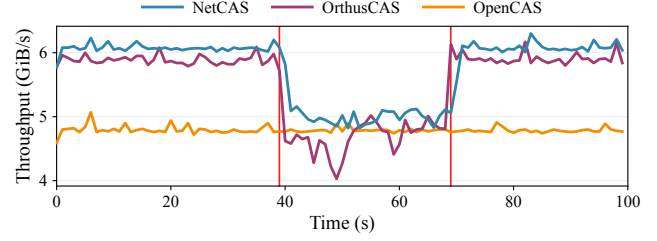


Fig. 12. Throughput under injected congestion (30s). Filebench seqread (Workload B) throughput over time for the three systems; the highlighted interval corresponds to 30 seconds of RDMA contention from 40 flows (two servers, 2.5Gbps cap per flow).

a precomputed Perf Profile that captures device-level scaling characteristics across workload configurations, (ii) host-local end-to-end monitoring of throughput and latency from the NVMe-oF completion path, and (iii) low-overhead split enforcement with batched weighted round robin. Together, these components allow each host to adapt quickly to time-varying conditions without depending on a centralized control plane or cross-node coordination protocol.

VII. ACKNOWLEDGEMENT

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (RS-2025-24534142), and by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) under the Artificial Intelligence Convergence Innovation Human Resources Development Program (IITP-2026-RS-2023-00255968). The authors would like to thank Sungkyunkwan University, Ajou University, and Cornelis Networks for their support.

REFERENCES

- [1] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson, “Strata: A cross media file system,” in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 460–477.
- [2] S. Zheng, M. Hoseinzadeh, and S. Swanson, “Ziggurat: A tiered file system for {Non-Volatile} main memories and disks,” in *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 2019, pp. 207–219.
- [3] Y. Zhan, H. Hu, X. Yang, Q. Cao, H. Jiang, S. Wang, and J. Yao, “Rethinking the {Request-to-IO} transformation process of file systems for full utilization of {High-Bandwidth}-{SSDs},” in *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, 2025, pp. 69–86.
- [4] Y. Ren, D. Domingo, J. Zhang, P. John, R. Pitchumani, S. Kashyap, and S. Kannan, “{PolyStore}: Exploiting combined capabilities of heterogeneous storage,” in *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, 2025, pp. 539–555.
- [5] K. Wu, Z. Guo, G. Hu, K. Tu, R. Alagappan, R. Sen, K. Park, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “The storage hierarchy is not a hierarchy: Optimizing caching on modern storage devices with orthus,” in *19th USENIX Conference on File and Storage Technologies (FAST 21)*, 2021, pp. 307–323.
- [6] Z. Lin, L. Xiang, J. Rao, and H. Lu, “{P2CACHE}: Exploring tiered memory for {In-Kernel} file systems caching,” in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023, pp. 801–815.

- [7] H. Chen, C. Ruan, C. Li, X. Ma, and Y. Xu, "{SpanDB}: A fast,{Cost-Effective},{LSM-tree} based {KV} store on hybrid storage," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*, 2021, pp. 17–32.
- [8] A. Fedorova, K. A. Smith, K. Bostic, S. LoVerso, M. Cahill, and A. Gorrod, "Writes hurt: Lessons in cache design for optane nvram," in *Proceedings of the 13th Symposium on Cloud Computing*, 2022, pp. 110–125.
- [9] A. Verbitski, A. Gupta, D. Saha, M. Brahmesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao, "Amazon aurora: Design considerations for high throughput cloud-native relational databases," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 1041–1052.
- [10] Microsoft, "Azure sql database hyperscale," <https://learn.microsoft.com/en-us/azure/azure-sql/database/service-tier-hyperscale>, 2025, accessed: 2025-06-30.
- [11] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing key-value stores with fast in-network caching," in *Proceedings of the 26th symposium on operating systems principles*, 2017, pp. 121–136.
- [12] J. Shu, R. Zhu, Y. Ma, G. Huang, H. Mei, X. Liu, and X. Jin, "Disaggregated raid storage in modern datacenters," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023, pp. 147–163.
- [13] Z. Guz, H. Li, A. Shayesteh, and V. Balakrishnan, "Performance characterization of nvme-over-fabrics storage disaggregation," *ACM Transactions on Storage (TOS)*, vol. 14, no. 4, pp. 1–18, 2018.
- [14] G. Haas and V. Leis, "What modern nvme storage can do, and how to exploit it: High-performance i/o for high-performance storage engines," *Proceedings of the VLDB Endowment*, vol. 16, no. 9, pp. 2090–2102, 2023.
- [15] Z. Lin, L. Cao, F. Ahmed, H. Lu, and P. Sharma, "When caching systems meet emerging storage devices: A case study," in *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*, 2023, pp. 37–43.
- [16] J. Liu, H. Hadian, Y. Wang, D. S. Berger, M. Nguyen, X. Jian, S. H. Noh, and H. Li, "Systematic cxl memory characterization and performance analysis at scale," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2025, pp. 1203–1217.
- [17] Compute Express Link Consortium, "CXL Specification Revision 4.0, Version 1.0," https://computeexpresslink.org/wp-content/uploads/2026/02/CXL-Specification_rev4p0_ver1p0_2026February26_clean_evalcopy_v2.pdf, 2026, accessed: 2026-03-27.
- [18] NVM Express, Inc., "Nvm express base specification," <https://nvmexpress.org/specification/nvm-express-base-specification/>, 2026, accessed: 2026-03-27.
- [19] —, "Nvm express over fabrics specification (historical revisions)," <https://nvmexpress.org/specification/nvme-of-specification/>, 2026, accessed: 2026-03-27.
- [20] —, "Nvm express over fabrics rdma transport specification," <https://nvmexpress.org/specification/rdma-transport-specification/>, 2026, accessed: 2026-03-27.
- [21] —, "Nvm express over fabrics tcp transport specification," <https://nvmexpress.org/specification/tcp-transport-specification/>, 2026, accessed: 2026-03-27.
- [22] Z. Guz, H. Li, A. Shayesteh, and V. Balakrishnan, "Nvme-over-fabrics performance characterization and the path to low-overhead flash disaggregation," in *Proceedings of the 10th ACM International Systems and Storage Conference*, 2017, pp. 1–9.
- [23] Y. Kang and M. Liu, "Understanding and profiling {NVMe-over-TCP} using ntprof," in *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, 2025, pp. 1117–1136.
- [24] Open CAS, "Open Cache Acceleration Software," <https://open-cas.com/>, accessed: 2025-09-17.
- [25] Q. Zhang, P. Bernstein, B. Chandramouli, J. Hu, and Y. Zheng, "Dds: Dpu-optimized disaggregated storage," *arXiv preprint arXiv:2407.13618*, 2024.
- [26] C. Petersen, "Introducing lightning: A flexible nvme jbof," in *Engineering at Meta*, 2016, <https://engineering.fb.com/2016/03/09/data-center-engineering/introducing-lightning-a-flexible-nvme-jbof/>.
- [27] J. Markussen, L. B. Kristiansen, H. K. Stensland, and P. Halvorsen, "Multi-host sharing of a single-function nvme device in a pcie cluster," in *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2024, pp. 1638–1645.
- [28] K. Winstein and H. Balakrishnan, "Tcp ex machina: Computer-generated congestion control," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. ACM, 2013, pp. 123–134.
- [29] W. Y. Liang, M. F. Chang, Y. L. Chen, and J. H. Wang, "Performance evaluation for dynamic voltage and frequency scaling using runtime performance counters," *Applied Mechanics and Materials*, vol. 284, pp. 2575–2579, 2013.
- [30] S. Ha, I. Rhee, and L. Xu, "Cubic: A new tcp-friendly high-speed tcp variant," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 64–74, 2008.
- [31] J. Mars and L. Tang, "Whare-map: Heterogeneity in "homogeneous" warehouse-scale computers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013, pp. 619–630.
- [32] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling memcache at Facebook," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013, pp. 385–398.
- [33] Meta, "Data center fabric," <https://engineering.fb.com/2014/11/14/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/>, accessed: 2026-02-01.
- [34] Google, "TPU VM / TPU pod architecture," <https://cloud.google.com/tpu/docs/system-architecture-tpu-vm>, accessed: 2026-02-01.
- [35] Microsoft, "Azure AI superfactory architecture," <https://azure.microsoft.com/en-us/blog/azure-ai-foundry-from-exploration-to-production-with-enterprise-grade-ai/>, accessed: 2026-02-01.
- [36] Y. Cheng, A. Anwar, and X. Duan, "Analyzing Alibaba's co-located datacenter workloads," in *Proc. IEEE International Conference on Big Data*, 2018, pp. 292–297.
- [37] D. Huye, Y. Shkuro, and R. R. Sambasivan, "Lifting the veil on {Meta's} microservice architecture: Analyses of topology and request workflows," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023, pp. 419–432.
- [38] W. Su, A. Dhanotia, C. Torres, J. Gandhi, N. Gholkar, S. Kanaujia, M. Naumov, K. Subramanian, V. Andrei, Y. Yuan *et al.*, "Dcperf: An open-source, battle-tested performance benchmark suite for datacenter workloads," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, 2025, pp. 1717–1730.
- [39] R. Koller, L. Marmol, S. Sundararaman, V. Tarasov, and E. Zadok, "Write policies for host-side flash caches," in *11th USENIX Conference on File and Storage Technologies (FAST 13)*, 2013, pp. 45–58.
- [40] V. Tarasov, E. Zadok, and S. Shepler, "Filebench: A flexible framework for file system benchmarking," *login: USENIX Magazine*, vol. 41, no. 1, pp. 6–12, 2016, <https://www.usenix.org/publications/login/spring2016/tarasov>.
- [41] Filebench Developers, "Filebench wiki," <https://github.com/filebench/filebench/wiki>, 2024, accessed: 2026-03-23.